Identifying Evolvability for Integration

L. Davis and Rose Gamble*

Software Engineering & Architecture Team
Department of Mathematical and Computer Sciences
University of Tulsa, 600 S. College Ave., Tulsa, OK 74104
{davisl,gamble}@utulsa.edu

Abstract. The seamless integration of commercial-off-the-shelf (COTS) components offers many benefits associated with reuse. Even with successful composite applications, unexpected interoperability conflicts can arise when COTS products are upgraded, new components are needed, and the application requirements change. Recent approaches to integration follow pattern-based design principles to construct integration *architecture* for the composite application. This integration architecture provides a design perspective for addressing the problematic interactions among components within the application environment. However, little attention has been paid to the evolvability of these architectures and their embedded functionality. In this paper, we discuss the need for design traceability based on the history of interoperability conflicts and resolution decisions that comprise the integration architecture. Additionally, we advocate that certain functional aspects of a pattern can be pinpointed to resolve a conflict. Combining these two aspects of integration architecture design, we illustrate that often evolution is possible with minimal changes to the integration solution.

1 Introduction

Integration of software components is a well-accepted approach to address the various issues associated with in-house development of complex systems. However, it is not always a simple process. Both industry and academia are developing techniques, methodologies, and products to alleviate the problems surrounding building composite applications. One central point to be considered is how the inclusion of COTS products impacts interoperability. There is no doubt that in most cases, COTS products offer well-tested, vendor-supported software that would be burdensome to build in-house. Though such in-house development is performed (often because a product won't integrate properly), it can significantly increase development cost, while at the same time decreasing reliability and support.

There are many reasons why integration is difficult. Most have to do with the behavioral expectations of the component and its application environment.

* Contact author. This research is sponsored in part by AFOSR (F49620-98-1-0217) and NSF (CCR-9988320).

J. Dean and A. Gravel (Eds.): ICCBSS 2002, LNCS 2255, pp. 65–75, 2002.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2002		**00-00-2002 to 00-00-2002**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Identifying Evolvability for Integration	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Tulsa,Department of Mathematical and Computer Sciences,600 S. College Avenue,Tulsa,OK,74104**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**11**	

Other reasons include a shortage of tools to aid professionals in an integration effort. They must often rely on instinct instead of a principled approach to build composite applications. As in all software development, there may be disagreement as to what requirements, components, or middleware choices are malleable. Misjudgment of requirements can lead to unexpected behavior or the use of unacceptable products.

Middleware products are being heavily marketed as complete solutions to all integration needs. They can be a perfect fit, integrating components smoothly. In other cases, however, vendor consultants are needed to train, assist, and/or configure the solution. Overall, there is still a great deal of guesswork and complexity in implementing middleware as it is usually considered after a failed integration attempt. Basic principles of requirements engineering and software design must be diligently followed to achieve seamless integration.

In general, patterns provide an implementation approach to problems in the form of repeatable solutions. Architectural and design patterns have been defined that underlie middleware frameworks. Some issues need to be addressed within patterns to facilitate their use for general component integration. Many viable patterns are not identified as integration patterns. Those that are often do not detail the interoperability conflicts that they resolve. Patterns also do not naturally identify closed box functionality, like COTS products, present in the implementation. Thus, there is a lack of direction in pinpointing patterns that are descriptive enough to assist in the implementation of a composite application.

Given that integration goes smoothly, evolution can generate additional component integration issues, while, at the same time, making others obsolete. These problems are magnified when a COTS product is part of an integrated application. In fact, COTS products are especially susceptible to evolution, including radical changes, due to the need to attain and keep a broad customer base.

Integration solutions should be altered minimally for the continued reuse benefits. To do this, it is necessary to know how and why an existing integration solution is impacted by evolution in order to design it more robustly. This requires an understanding of why a pattern is use, how it can change and its effect on integration.

In this paper, we use the history of interoperability conflicts and resolution decisions that comprise the integration architecture as a basis for understanding the design. We advocate that certain aspects of a pattern can be pinpointed to resolve a conflict. Combining these two aspects of integration architecture design, we illustrate that often evolution is possible with minimal changes to the integration solution.

2 Background

Maintaining a high-level of abstraction at which to relate a system design remains the basis for describing an architecture for software. Through *software architecture* a system's computational elements, their interactions, and structural constraints can be expressed at a high, yet meaningful, level of abstraction

[1]. Conceptual system issues can then be explained, discussed and modeled without becoming entangled in implementation and deployment details. Characteristics defined with respect to architectural styles include those that describe the various types of components and connectors, data issues, control issues, and control/data interaction issues [2,3,4].

Connectors have become increasingly important in software architecture analysis, forming the basis for component connection [5,6,7,8]. Explicit descriptions of connectors are desirable as these can allow for design choices among and analysis of existing interaction schemes, along with the specification of new connectors. This effort can allow designers the flexibility to choose the correct interoperability schemes in an integrated application.

Architectures and patterns that form middleware frameworks provide guidance to "in-house" integration efforts. Common patterns considered as integration patterns are the Proxy [10] and Broker that afford clients and servers location transparency, plug and play mobility, and portability [10,11]. Recent integration patterns, including the Wrapper-Façade, Component Configurator, and the Interceptor, represent repeatable solutions to particular integration problems [12]. Enterprise Application Integration patterns, such as the Integration Mediator [13] focus on integrating enterprise application components. These patterns provide functionality such as interface unification/reusability, location transparency/negotiation, and decoupling.

The most salient point to be made concerning integration patterns is their lack of connection between the interoperability problems requiring the use of the pattern and the reason the pattern resolves those problems. As more patterns are defined and their complexity increases, at stake becomes the understandability of the integration pattern and the history of decisions for its use. In short, in their current state, patterns do not focus on issues of integration solution evolution.

The evolutionary properties of components can be captured using principles of software architecture. One goal is to find an architecture to which other architectures can transition easily [1]. Though architecture migration is a plausible solution, it is not always feasible for all systems, especially COTS products where many properties are hidden. In a similar vein, researchers examine constraints on reconfiguring architectures to assess their response to evolution [15]. This illustrates that certain properties of components and connectors lessen the impact of change [16]. Certain architecture description languages support architecture-based evolution through the changes in topology [16], optionality [17], and variability [18].

Analysis methods exist to assess the evolvability and reusability of a component architecture. One method employs a framework and a set of architectural views [19]. The architectural views include information gathered during various system lifecycles as well as stakeholder requirements. Our research relies on static property analysis methods to evaluate characteristics of the architecture in an effort to identify potential interoperability problems [20,21].

3 Fundamentals for Understanding Integration Solution Design

Collidescope is a prototype assessment tool to provide developers with a means for evaluating integration solution design decisions [22]. The ultimate goal is to provide the developer with a visible link between interoperability problems, their causes, and their solutions. Collidescope currently implements the foundational underpinnings needed to determine potential *problematic architecture interactions* (PAIs). PAIs are defined as interoperability conflicts that are predicted through the comparison of relevant architecture interaction characteristics and require intervention via external services for their resolution [22]. Utilizing broad, but descriptive, architectural characteristics of the component systems provides dual benefits. They aid discovery of PAIs and the assessment of inevitable future conflicts brought on by evolution [23]. In fact, Collidescope does not require all component characteristics to have values. It can work with only a partial set. This feature is very important because little information may be known about a COTS product, especially one with which a developer has had little experience. Of course, some potential PAIs may be missed.

Figure 1 depicts two linked components, Alpha and Beta, in a composite application. Control structure, the structure that governs the execution in the system, is identified for each of the components. Alpha has a concurrent control structure, while Beta's is single-threaded. The application has values for many of the same characteristics as the component with a different granularity. For instance, the control structure of the application governs how the *components* coordinate their execution. We will return to application characteristics in Sect. 4.

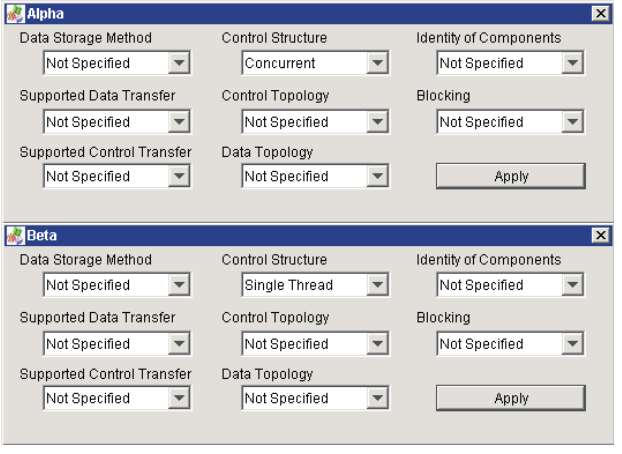

Fig. 1. Architecture Characteristics with Values for Control Structure

As a first pass analysis, Collidescope detects thirteen PAIs in one of three categories. These categories represent distinct issues that arise in component communication: expectations for data transfer, expectations for control transfer, and how interaction between components is initialized.

Two types of static analysis are performed [20]. The first is *component-component* analysis in which component characteristic values are compared. For *application-component* analysis, the application characteristics are compared with each component. Figure 2 displays the potential PAIs found by a component-component analysis using Alpha and Beta. In addition, we introduce an application, Omega, for application-component analysis. The name is in bold. The characteristic is in italics followed by a value. The number (1-13) is the conflict preceded by its category.

When comparing these characteristics it becomes apparent that communication between concurrent and single-threaded components will be difficult as single threaded components expect directed control transfer and block upon initiation. Concurrent components, on the other hand, run despite the execution state of other participating components.

Alpha			**Beta**	
Control Structure	Concurrent		Control Structure	Single Thread
4	Control Transfer	Sequencing multiple control transfers		
Omega			**Alpha**	
Control Structure	Single Thread		Control Structure	Concurrent
4	Control Transfer	Sequencing multiple control transfers		
Omega			**Alpha**	
Control Topology	Hierarchical		Control Structure	Concurrent
1	Control Transfer	Restricted points of control transfer		
Omega			**Alpha**	
Data Topology	Hierarchical		Control Structure	Concurrent
5	Data Transfer	Restricted points of data transfer		
10	Data Transfer	Sequencing multiple data transfers		

Fig. 2. Problematic Architecture Interactions

The next step in the assessment is the identification of integration elements that resolve the PAIs. This completes the design path from an identified problem to the fundamental solution. We model the basic functionality needed for integration as three integration elements: *translator, controller,* and *extender* [24,25]. These integration elements supplement the traditional architecture connectors.

A *translator* has some basic properties. It should communicate with other components independent of their identities. Input must have a uniform structure that is known by the translator's domain. Third, conversions must be represented by a total mathematical relation or composition of relations that maps the input to the output. Translators are particularly necessary in heterogeneous, COTS-based integrations, as consistently formatted data is never expected between vendors.

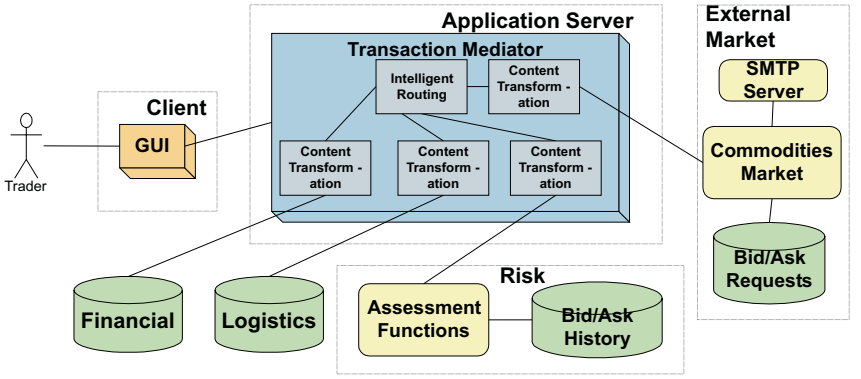

Fig. 3. The InfoTrade System Architecture

A *controller* integration element coordinates and mediates the movement of information between components using predefined decision-making processes. The decisions include determining what data to pass, from which component to accept data, and to which component data should be sent. Multiple decisions can be made within a single controller. Decisions can be based upon input data, input components, output components, or a combination of data and components.

An *extender* integration element adds those features and functionality to an integration solution to further adapt it to the application environment, embodying those behaviors not performed by a translator or controller (e.g., buffering, adding call-backs, opening files, and performing security checks). Because of the diverse behavior of extenders, each distinct action is modeled independently.

The above integration elements may be combined with each other and with simple connectors (e.g., a UNIX pipe) to form *integration architectures* as needed to resolve specific conflicts. An integration architecture, then, is defined to be the software architecture description of a solution to interoperability problems between at least two interacting component systems [11,26]. An integration architecture forms the foundation of design patterns and off-the-shelf (OTS) middleware.

The use of static, relevant architectural characteristics, standardized conflicts, and simple integration functions to resolve conflicts provides the history of design decisions. Therefore, when evidence changes, it points directly to the resulting functionality that is affected.

4 An Evolving Application

In this section, we study the requirements of a composite application called InfoTrade, a system for automated trading of commodities, like oil and gas. The drive to develop such an application comes at a time when automated stock trading is in full swing. Yet, commodities trading is just beginning to be automated. Many of the needed independent software components (some

of which are COTS) are in place but only partial automated trading is being done. Much of the information before and after a trade is entered manually. Furthermore, because automated trading is limited, there is a mix of old and new styles of information and flow that must be blended.

The InfoTrade system architecture is shown in Fig. 3. The participating components include External Market, Risk, Logistics, and Financial. The External Market is comprised of a national commodities market information module, a database to house all of the bid/ask requests put to the market, and a SMTP for dynamic messaging. Risk uses a history of Bid/Ask requests to assess the financial exposure of the corporation. Logistics is a database of transportation (car, boat, plane, etc.) information, including specifications and statistics associated with transporting commodities. Financial is a database of customer billing information, as well as, corporate-wide financial information.

InfoTrade integrates COTS components essential to the execution of a commodities trade using the Transaction Mediator (Fig. 3) – an implementation of the Integration Mediator architecture [13]. This solution accommodates the different component data formats and communication methods. Within the integrated application, the Transaction Mediator is stateless, needing only the current Bid/Ask request to perform its mediation. Thus, to make a trade in this system, the user places a bid, which is routed through the Transaction Mediator. The Intelligent Router coordinates the direction of the Bid/Ask message. Content Transformers intercept the message and transform it to a format for their respective components. Risk analysis is performed only on a weekly basis when the market is closed and no bids are being placed. The Intelligent Router calls the External Market to request the current store of Bid/Ask requests, sequencing any concurrent communications such as a SMTP packet being sent. It then routes these records to Risk for financial exposure analysis. During communications to Risk and External Market, a unique Content Transformer translates incoming/outgoing requests to ensure the request data formats are correct.

With the expansion of e-commerce opportunities for service-oriented corporations, more companies desire an automated commodities facility with *real-time risk analysis*. This institutes a new composite application requirement causing the application architecture characteristics to evolve.

The real-time requirement places new demands on the Transaction Mediator, as it must retain the current Bid/Ask request to arbitrate the concurrently executing trade and the financial exposure calculation. The question becomes how can the integration solution now meet this new requirement? One alternative is to scrap the existing implementation, choosing more dynamic integration architectures such as a Broker. Another alternative is to re-write the Transaction Mediator, perhaps transitioning from its combined Java and JMS implementation to one utilizing real-time CORBA. Both of these choices go against the reuse ideals that led to the original integration.

The design decisions supported by our methodology (Sect. 3) provides a direct link from the application requirements to the integration elements that make up the integration solution. This fosters evolution by providing the developer with insight into which parts of the integration solution should be modified or replaced, as well as insight into where additional pieces of functionality are

needed. The assessment method does not discount any of the alternatives, but helps to determine which – a new design, re-implementation, or evolution – is the best choice

There are many factors that contribute to the formation and, subsequent, evolution of the InfoTrade integrated application. First, it is important to look at the components in the current application. As these are COTS systems, little is known of their internal functionality. Besides their obviously dissimilar data formats, only the component-level characteristic *control structure* can be discerned. Refer to Alpha and Beta in Fig. 1 as the External Market and Risk components, respectively.

For InfoTrade to accommodate real-time risk analysis, the application itself must be characterized differently. Table 1 shows how this change affects how the application will be newly described. The component values remain the same.

Table 1. Evolving Application Characteristics

Characteristic	**Original Value**	**Evolved Value**
Control Structure	Single -Thread	Concurrent
Control Topology	Hierarchical	Arbitrary
Data Topology	Hierarchical	Arbitrary
Synchronization	Synchronous	Asynchronous

The way in which changing application characteristics shape the current integration can be seen in a comparison of these new values to the current control structure values of the components. Collidescope detects the new PAIs shown in Fig. 4.

Gamma		**Beta**	
Control Structure	Concurrent	Control Structure	Single Thread
4	Control Transfer	Sequencing multiple control transfers	
Gamma		**Beta**	
Data Topology	Arbitrary	Control Structure	Single Thread
3	Control Transfer	Inhibited rendezvous	
5	Data Transfer	Restricted points of data transfer	
Gamma		**Beta**	
Control Topology	Arbitrary	Control Structure	Single Thread
1	Control Transfer	Restricted points of control transfer	
3	Control Transfer	Inhibited rendezvous	
Gamma		**Alpha**	
Control Structure	Concurrent	Control Structure	Concurrent
4	Control Transfer	Sequencing multiple control transfers	

Fig. 4. The PAIs Resulting from the Evolved Application Characteristics

The stateless Transaction Mediator (depicted in Fig. 3) as initially implemented only embodies translation and control. In the new application, both are still needed. However, the Transaction Mediator now must also buffer the data being processed by Risk and External Market in order to calculate financial exposure while placing a bid. Otherwise, refusal of a trade by Risk will result in an *inhibited rendezvous* as that refusal cannot be correlated with the actual correct request to circumvent the acceptance of the trade by External Market. The PAIs in Fig. 4 reflect this problem as well as additional conflicts that the Transaction Mediator currently handles. Figure 5 shows the evolved architecture.

By using this style of analysis, a developer is more likely to ascertain the distinctive problems that are caused by changing characteristics. Moreover, a direct and minimal resolution may be achieved.

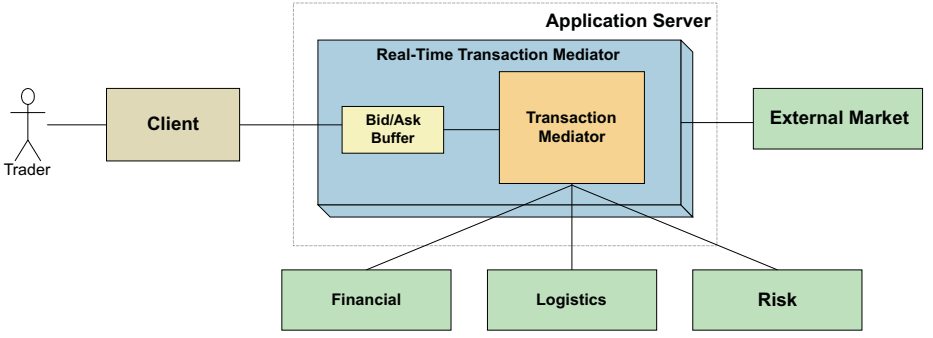

Fig. 5. The New Architecture of InfoTrade

5 Conclusions

Little attention has been paid to the evolvability of these architectures and their embedded functionality. In this paper, we show how design choices rely on the history of interoperability conflicts and resolution decisions that comprise the integration architecture. Additionally, we advocate that certain functional aspects of a pattern can be pinpointed to resolve a conflict. Combining these two facets of integration architecture design, we illustrate that often evolution is possible with minimal changes to the integration solution.

The approach we advocate has both advantages and limitations. The assessment, though a first-pass, is at a high-level of abstraction, and forms a reliable history of design information. In turn, the history is easily maintainable. Given the high-level of abstraction present in the assessment, evolutionary impacts are relatively easy to determine. However, the abstraction level restricts the depth of the assessment, as exact implementation details are not provided. Furthermore, we have not yet proven the analysis is scalable to either applications comprised of a large number of components or applications with diverse middleware products in use. We reserve these findings for future work.

References

1. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs, NJ: Prentice Hall, (1996).
2. Abd-Allah, A.: Composing Heterogeneous Software Architectures. Ph. D. Dissertation, Computer Science, University of Southern California, (1996).
3. Garlan, D., Allen, A., Ockerbloom, J.: Architectural Mismatch, or Why it is hard to build systems out of existing parts. In, 17^{th} International Conference on Software Engineering. Seattle, WA, (1995).
4. Shaw, M., Clements, P.: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, 1^{st} International Computer Software and Applications Conference. Washington, D.C., 6-17, (1997).
5. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodologies, 6(3): 213-49, (1997).
6. Garlan, D.: Higher-Order Connectors, Workshop on Compositional Software Architectures. Monterey, CA, January 6-7, (1998).
7. Keshav, R., Gamble, R.: Towards a Taxonomy of Architecture Integration Strategies. 3^{rd} International Software Architecture Workshop, 1-2, (1998).
8. Medvidovic, N., Gamble, R., Rosenblum, D.: Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms. 4^{th} International Software Architecture Workshop, (2000).
9. Mehta, N., Medvidovic, N., Phadke, S.: Towards a Taxonomy of Software Connectors. In, 22^{nd} International Conference on Software Engineering, (2000).
10. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, (1996).
11. Mularz. D.: Pattern-based integration architectures. PloP, 1994.
12. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley & Sons, (2000).
13. Lutz, J. C.: EAI Architecture Patterns. EAI Journal, (2000).
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, (1995).
15. van der Hoek, A., Heimbigner, D., Wolf, A.: Capturing Architectural Configurability: Variants, Options, and Evolution. Technical Report CU-CS-895-99, Department of Computer Science, University of Colorado, Boulder, Colorado, December (1999).
16. Oreizy, P., Medvidovic, N., and Taylor, R.: Architecture-Based Runtime Software Evolution. In Proceedings of the 20^{th} International Conference on Software Engineering, (1998), 177-186.
17. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model For Consumer Electronics Software. IEEE Computer, Vol. 33, No. 3, (2000), pp. 78-85.
18. Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G.: Abstractions For Software Architecture And Tools To Support Them. IEEE Transactions on Software Engineering 21(4), (1995), 314-335.
19. Lung, C.-H., Bot, S., Kalaichelvan, K., Kazman, R.: An Approach to Software Architecture Analysis for Evolution and Reusability. Proceedings of CASCON '97, (Toronto, ON), (1997).
20. Davis, L., Gamble, R., Payton, J., Jonsdottir, G., Underwood, D.: A Notation for Problematic Architecture Interactions. In Foundations of Software Engineering '01, (2001).

21. Kazman, R., Klein, M., Clements, P.: ATAM: Method for Architecture Evaluation. Carnegie Mellon University, (2000).
22. Payton, J., Davis, L., Underwood, D., Gamble, R.: Using XML for an Architecture Interaction Conspectus. In XML Technologies and Software Engineering (2001).
23. Davis, L., Gamble, R., Payton, J.: The Impact of Component Architectures on Interoperability. Journal of Systems and Software, (to appear 2002).
24. Keshav, R.: Architecture Integration Elements: Connectors that Form Middleware. M.S. Thesis, Dept. of Mathematical & Computer Sciences: University of Tulsa, (1999).
25. Keshav, R., Gamble, R.: Towards a Taxonomy of Architecture Integration Strategies. 3^{rd} International Software Architecture Workshop, 1-2, (1998).
26. Payton, J., Gamble, R., Kimsen, S., Davis, L.: The Opportunity for Formal Models of Integration. In, 2^{nd} International Conference on Information Reuse and Integration, (2000).